Languages

Definition of Languages

In this section we will discuss the formal definition of "languages", give a number of examples, and discuss some standard constructions.

A **language** in a given alphabet is simply a set of strings in that alphabet.

So a language is nothing more than a separation of the strings in that alphabet in two groups: Those that "belong" to this language and those that don't. Let's think of some examples:

1. Alphabet all decimal digits. We can talk about the language of "valid numbers". It will consist of all strings (sequences of digits) where there is always at least one digit and either it's the digit 0 by itself or the first digit in the sequence is non-zero.
2. Together with that language, we can also think of its "complement": All "non-valid numbers". This consists of the empty string, as well as any string that starts with a 0 and has at least one more digit.
3. Alphabet all letters in the english language, and a hyphen. The language of "valid words" consists of the "finite" list of all those strings that are valid words in the english language, as for example described by a dictionary.
4. Alphabet all letters. We can consider the language of all "keywords in OCAML", a finite list of words that are reserved words in the OCAML language, as provided by the language's manual.
5. Alphabet all letters plus numbers and a few special symbols. The language of "valid identifiers" consists of all strings that can be used as an identifier/variable in a programming language like OCAML. These typically require that the first character is a letter, but that subsequent characters could be anything, including letters, numbers, dots, dashes depending on what the programming language definition allows. OCAML for example allows any number single quote characters but only if they are at the end of the string.

It is worth pausing here for a moment, and thinking about how you could specify the kinds of languages we are describing above in say OCAML. What kind of structure could we use? What would be its advantages and disadvantages?

After a bit of thought you'll probably arrive at two different descriptions, both of which really end up describing ways of describing a "set":

1. Using a "predicate", i.e. a function that takes in a string from the alphabet and returns a boolean stating whether that particular string is in the language/set. This is probably the easiest description. But it offers us no way to actually exhibit elements that belong to the language.

2. Listing all the strings in the language. If a language is finite this is easy to do. If it is not, we could maybe specify it via an infinite "stream", that on demand can be called on to give us the "next" element in the language. This makes it easy to exhibit elements that belong to the language, but it makes it rather hard to determine if a specific string is part of the language or not.

So basically:

> We can "specify" a language by providing either: 1. A way to tell, for a given string, if it belongs in the language or not, or 2. A way to generate strings from the language, eventually *all* the strings from the language.

Here is a question worth thinking about for a moment, or perhaps longer:

> Given descriptions for two languages, via either of the two methods above, can we tell if those descriptions actually correspond to the same language? In other words, can we tell when two languages are "equal"?

Standard Operations on Languages

There are a number of standard operations on languages, that allow us to create new languages from existing ones:

Union If A and B are two languages on the same alphabet, thought of as sets, then their **union** $A \cup B$ consists of all strings that belong to at least one of the two languages. so:

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$$

Intersection If A and B are two languages on the same alphabet, thought of as sets, then their **intersection** $A \cup B$ consists of all strings, if any, that belong to both languages:

$$A \cap B = \{x \mid x \in A \text{ and } x \in B\}$$

Complement If A is a language, its **complement**, usually denoted A' or A^c, consists of all strings in the alphabet that are not in the language:

$$A' = \{x \mid x \notin A\}$$

Concatenation If A and B are languages, their **concatenation** consists of all strings formed by concatenating together a string from A and a string from B:

$$AB = \{ab \mid a \in A \text{ and } b \in B\}$$

Kleene Star This is an operation related to languages, that you do not often encounter in sets in general. The **Kleene star** of a language, denoted by A^*, is formed by taking sequences of strings from the language:

$$A = \{a_1 a_2 \cdots a_n \mid a_i \in A,\, n \geq 0\}$$

Note in particular that the empty string is always part of the Kleene Star of any language.

Here is a challenge problem. Some parts of it are easier than others and will be part of your homework.

Challenge

Given one of the language descriptions above (as a predicate or as a stream), can you describe how to obtain descriptions for each of the language constructs just described? For example, could you instruct a computer, if they are given predicates for languages A and B, to create predicates for Union/Intersection/Complement/Concatenation?

In a certain sense, talking about languages amounts to talking about computation. For instance, saying that I have implemented a way to compute, given a number, whether that number is prime or not is exactly the same as saying that I have written a program that can "recognize" the language:

$$A = \{x \mid \text{ the number represented by } x \text{ is prime } \}$$

Pretty much most problems in computing can be rephrased in such a manner. Therefore being able to recognize languages, i.e. being able in a mechanistic way to determine if a string is in a language or not, is the pivotal question we will be attempting to answer in this course.